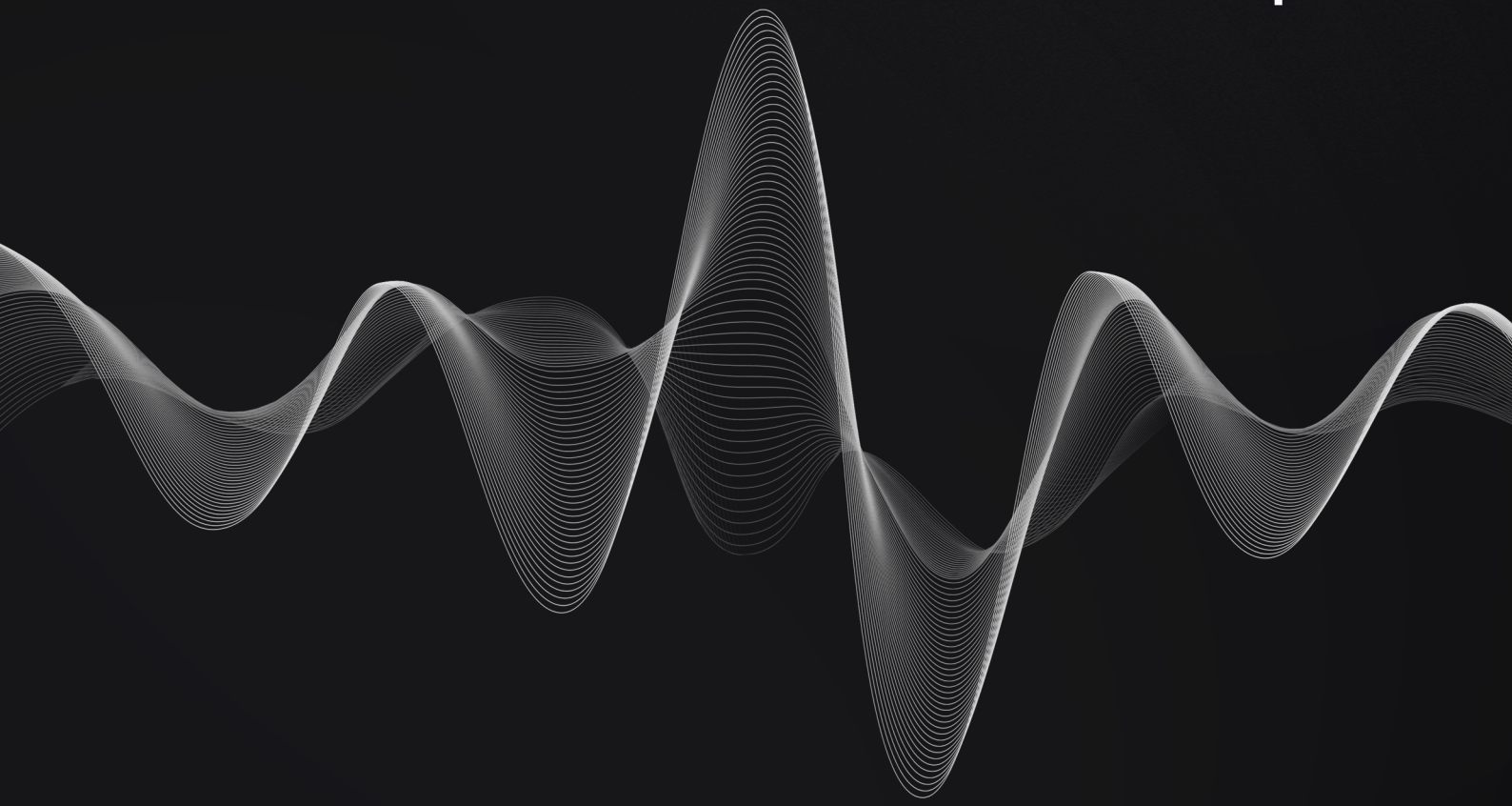
RESONANCE

Velvet Capital

Velvet Capital

V3 Smart Contracts Audit Report



Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

Document Control

PUBLIC **FINAL**(v2.1)

Audit_Report_VLVT-V3C_FINAL_21

May 3, 2024	v0.1	Michał Bajor: Initial draft
May 10, 2024	v0.2	Michał Bajor: Added findings
May 13, 2024	v0.3	João Simões: Added findings
May 14, 2024	v1.0	Charles Dray: Approved
May 30, 2024	v1.1	João Simões: Reviewed findings
Jun 6, 2024	v2.0	Charles Dray: Finalized
Sep 19, 2024	v2.1	Charles Dray: Published

Points of Contact	Vasily Nikonov	Velvet Capital	vasily@velvet.capital
	Charles Dray	Resonance	charles@resonance.security
Testing Team	Michal Bajor	Resonance	michal.bajor@resonance.security
	João Simões	Resonance	joao.simoes@resonance.security
	Ilan Abitbol	Resonance	ilan.abitbol@resonance.security
	Michał Bazyli	Resonance	michal.bazyli@resonance.security

Copyright and Disclaimer

Contents

Executive Summary

Velvet Capital contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between April 25, 2024 and May 15, 2024. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 21 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

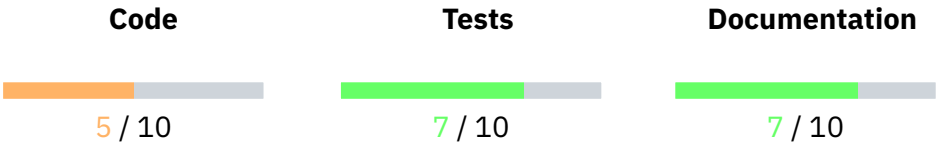
The ultimate goal of the audit was to provide Velvet Capital with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.

System Overview

Velvet Capital is a protocol that facilitates the management of users' crypto portfolios. A so called Intent Operating System for DeFi makes it easy for users to control what they do with their assets. One of the key features is an implicit batching of operation allowing for quick portfolio transitions via on-chain swapping. Velvet Capital exposes users to ready-to-use vaults while also allowing them to create their own.

Velvet Capital brings together the latest research and allows users to use the best fund managers and algorithms to manage their assets if they choose to. The capital management is not only oriented around keeping and trading assets, but it also involves leveraging yield-farming to maximize capital efficiency. Subsequent integrations with real-world assets, derivative based strategies and managing assets on multiple chains are planned to be implemented in the future.

Repository Coverage and Quality



Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows some development best practices and makes use of some standard libraries, but does not make use of known patterns and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is average**.

- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is 80%. Overall, **tests coverage and quality is good**.
- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is good**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: `Velvet-Capital/v3-contract/contracts`
- Hash: 58c8e14e10c7f5533557452dc5b9968e55444fa1

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- **"Quick Win"** Requires little work for a high impact on risk reduction.
- **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	Portfolio Draining Via Unprotected Call With Potential Token Theft In The Future		Resolved
RES-02	Possibility Of Locking Ether In Contract		Resolved
RES-03	Missing Access Control On _authorizeUpgrade()		Resolved
RES-04	Missing Validation Of Emergency During Protocol Unpause		Resolved
RES-05	Insufficient Validation Of Enabled Tokens		Acknowledged
RES-06	Missing _disableInitializers() On Upgradeable Contracts		Resolved
RES-07	Oracle Might Return Stale Results		Acknowledged
RES-08	Missing Validation Of transferable And transferableToPublic During Contract Deployment		Resolved
RES-09	Incorrect Usage Of Initializing Functions		Acknowledged
RES-10	Potentially Invalid Logic In Role Transfer Function		Acknowledged
RES-11	Excessive Fee Configuration is Possible for Protocol Fee Settings		Resolved
RES-12	Lack of ERC165 Checks		Acknowledged
RES-13	Redundant Checks On latestRoundData()		Resolved

RES-14	Unused Functions		Resolved
RES-15	Price Check Not Taking Into Account The Return Type		Resolved
RES-16	Unused Input Parameter _protocolConfig		Resolved
RES-17	Redundant Checks On updateTokens()		Resolved
RES-18	Redundant Checks On multiTokenWithdrawal()		Resolved
RES-19	Transfer Of Super Admin Ownership Does Not Revoke Other Roles		Acknowledged
RES-20	Missing Length Checks		Resolved
RES-21	Possible Accidental Event Flooding		Resolved

Portfolio Draining Via Unprotected Call With Potential Token Theft In The Future

Critical **RES-VLVT-V3C01** Access Control **Resolved**

Code Section

- contracts/rebalance/Rebalancing.sol#L114-138

Description

The `Rebalancing` contract allows the Asset Manager to influence the `Portfolio`'s configuration, namely the distribution of tokens inside of the portfolio. Additionally, it is also possible to use the `updateTokens` function to update the tokens themselves that a given `Portfolio` consists of. It was observed that the `updateTokens` function is publicly callable by anyone, not just by the Asset Manager. This fact, along with other minor issues, leads to draining the `Portfolio` contract from its assets.

The whole `updateTokens` flow consists of:

1. Checking if supposed new tokens are whitelisted (if whitelisting is enabled).
2. Checking if the number of new tokens exceeds the asset number limit.
3. Updating the token weights, by pulling previous tokens from `Portfolio`'s vault and selling them using the swap target (via `Enso` protocol).
4. Ensuring that the tokens were swapped for the expected price (done in the `EnsoHandler` contract).
5. Ensuring that all of the tokens pulled from `Portfolio`'s vault were sold, i.e. the `EnsoHandler` contract has a balance of `0` for each token present previously in the `Portfolio`'s vault.
6. Ensuring that the `Portfolio`'s vault has a positive balance of each new token.
7. Ensuring that the `Portfolios` vault has a zero balance for each token that was removed from the `Portfolio`.

The flow also requires that a legitimate, Velvet-deployed handler contract is used in the process, but the scenario described below uses the legitimate handler so this requirement was omitted.

The input parameters for the `updateTokens` function are not checked if they are internally consistent. Specifically, the tokens used by `EnsoHandler` contract to ensure successful swap can be different than the actual tokens received from the swap operation.

If token whitelisting is disabled, then the malicious user might create own token contracts (or any contract that will implement `balanceOf` and `transfer` function with accordance to `ERC20` standard). Providing those contracts' addresses as `_newTokens` and as `tokens` inside of the `_callData` in the `updateTokens` function will allow the malicious user to control the checks from points `4` and `6` of the flow described above.

Furthermore, the `updateTokens` caller also controls the `callDataEnso` argument, which is essentially an encoded call to the `Enso` protocol that dictates the swapping. Hence, the attacker can use it to instruct the `Enso` protocol to sell all of the tokens previously held in the `Portfolio` for any legitimate other token supported by the swap. This will assure that checks from points `5` and `7` are fulfilled.

If token whitelisting is enabled, then a malicious user needs to use the legitimate token that is whitelisted. Before calling the `updateTokens` malicious user would need to transfer some tokens (even `1`) to make the `Portfolio`'s vault balance non-zero for those whitelisted tokens. That way, the check from point `6` is fulfilled. Those legitimate tokens' addresses would need to be provided as `_newTokens`, and custom malicious contract would be provided as `tokens` embedded in the `_callData` variable.

As a consequence, a malicious user can force the protocol to sell all of the legitimate tokens and essentially lock them in the handler contract causing severe capital loss.

Furthermore, the `Enso` protocol's contract used by the Velvet's swap handler was investigated. It appears that currently it does not support swapping via `delegatecall`. However, should this feature be enabled, then this finding would result in a token theft, as the attacker could use the `callDataEnso` to initiate a `delegatecall` to the contract that he controls. As the `delegatecall` mechanism preserves the values for `msg.sender` then the token transfer could be initiated to steal the tokens instead of swapping them and locking the swapped for tokens inside of the handler contract.

Recommendation

It is recommended to first implement the proper access control mechanism, so that the `updateTokens` function can also be called only by the Asset Manager. Furthermore, additional checks that would assure the input data's internal consistency is strongly suggested. Namely, the `_newTokens` and `tokens` variables along with the swap targets embedded in `callDataEnso` variable need to be the same.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Possibility Of Locking Ether In Contract

High | RES-VLVT-V3C02 | Business Logic | Resolved

Code Section

- contracts/core/management/TokenExclusionManager.sol#L257
- contracts/core/Portfolio.sol#L82
- contracts/handler/ExternalSwapHandler/EnsoHandler.sol#L75
- contracts/handler/ExternalSwapHandler/EnsoHandlerBundled.sol#L67
- contracts/rebalance/Rebalancing.sol#L216

Description

The `Portfolio` contract serves as the primary entry-point for users to interact with the protocol. Among other functionalities, it defines a `receive` endpoint that is executed whenever a call with empty `calldata` is received. The `Portfolio` contract will accept any form of regular `Ether` transfers. However, it was observed that it does not implement any way for user to receive `Ether` from the contract. As a consequence, any `Ether` transfer to the `Portfolio` contract will result in a permanent asset lost for the user.

Similarly, the same `receive` function is implemented by the `EnsoHandler` and `EnsoHandlerBundled` contracts, which are used as a intermediary between the Velvet protocol and Enso protocol, by the `Rebalancing` contract and by the `TokenExclusionManager`.

Recommendation

It is recommended to remove the `receive` function from the `Portfolio`, `EnsoHandler`, `EnsoHandlerBundled`, `Rebalancing` and `TokenExclusionManager` contracts.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Missing Access Control On _authorizeUpgrade()

High | RES-VLVT-V3C03 | Access Control | Resolved

Code Section

- contracts/core/management/TokenExclusionManager.sol#L259-L261

Description

The function `_authorizeUpgrade()` is used in OpenZeppelin's Upgradeability design patterns to provide developers a possibility of enforcing access control during upgrades of the smart contracts. By default, smart contracts that inherit this upgradeability design can be upgraded by any user without any access control implemented.

The usage of the function `_authorizeUpgrade()` on the protocol does not enforce proper access control on which specific users may upgrade the smart contract. As such, any malicious user may use this opportunity to upgrade the smart contract with malicious code and ultimately be able to compromise and takeover the entire protocol.

Recommendation

It is recommended to implement the necessary access control mechanisms for upgrading smart contracts. OpenZeppelin's `AccessControl` or `Ownable` contracts may be used to achieve this purpose.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Missing Validation Of Emergency During Protocol Unpause

Medium | RES-VLVT-V3C04 | Data Validation | Resolved

Code Section

- contracts/config/protocol/SystemSettings.sol#L51-L54

Description

The function `setProtocolPause()` does not validate if the protocol is in emergency pause before unpausing. This means that it is possible for the protocol to be incorrectly unpaused at will, which will trigger many instances of undefined behaviour across the protocol where certain functionalities only check for either `isProtocolPaused` or `isProtocolEmergencyPaused` and not both.

Recommendation

It is recommended to implement a validation to check if the protocol can be unpaused during an emergency pause. This should be done to guarantee consistency between protocol and emergency pauses and unpauses.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Insufficient Validation Of Enabled Tokens

Medium | RES-VLVT-V3C05 | Data Validation | Acknowledged

Code Section

- contracts/config/protocol/TokenManagement.sol#L19

Description

When initialized, updated, or used, the vault tokens are not verified whether they are enabled or not by the `TokenManagement` smart contract and its function `isTokenEnabled()`. The only instance where it is being validated is within the function `getVaultValueInUSD()` which is only called when the Asset Manager calls the function `chargePerformanceFee()`. This means that every interaction with vault tokens on the platform is not tracked and verified whether or not the tokens are enabled.

Recommendation

It is recommended to implement verifications during interactions with the vault tokens that will ensure the tokens are enabled and available for use.

Status

The issue was acknowledged by Velvet's team. The development team stated "The check for whether the tokens are enabled is only required when using the price oracle to ensure we have a price feed for the specific token. When enabling tokens, we don't want to restrict asset managers from using any tokens. Users have the option to invest in portfolios with only whitelisted tokens. The token list can't be updated by the asset manager, so users will know what to expect.".

Missing _disableInitializers() On Upgradeable Contracts

Medium | RES-VLVT-V3C06 | Business Logic | Resolved

Code Section

Not specified.

Description

An implementation contract should not be left uninitialized. An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy.

Several implementation instances across the protocol do not disable contract initialization with the usage of the function `_disableInitializaers()`.

Recommendation

It is recommended to prevent the initialization of the implementation contract when upgradeable/proxy contracts are being used. This can be done with the following code snippet:

```
constructor() {
    _disableInitializers();
}
```

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Oracle Might Return Stale Results

Low | **RES-VLVT-V3C07** | Data Validation | **Acknowledged**

Code Section

- contracts/oracle/PriceOracle.sol#L31

Description

The `PriceOracle` contract received a price for a given asset using Chainlink oracle's `latestRoundData` function. Then, it then assures that the price is not expired using the `updatedAt` value returned from Chainlink's oracle. However, the time threshold used to verify the data staleness is set to `25 hours` and is used for every asset. It must be noted that the different assets in Chainlink's infrastructure have different heartbeat intervals. Assets used more often have the interval of `1 hour`, while less commonly used ones can have the interval of `24 hours`. Using a `25 hour` threshold for assets with smaller heartbeat interval will make the Velvet's Oracle accept the price even if it is stale. If such price would be accepted, it would make the swap calculations incorrect and might result in a loss of funds for protocol, and in consequence for the users.

The heartbeat intervals are known for each asset and they do not change often during normal operation.

Recommendation

It is recommended to use a separate expiration threshold value for each asset. Each threshold should not exceed the heartbeat interval of a particular asset it is related to. Such a threshold should only be possible to set by an administrator and only during configuration stage (i.e. whenever a new token is added) or whenever that heartbeat value would change.

Status

The issue was acknowledged by Velvet's team. The development team stated "Having a threshold for each pair would lead to high maintenance and the transaction might fail if we don't update it fast enough".

Missing Validation Of transferable And transferableToPublic During Contract Deployment

Low | RES-VLVT-V3C08 | Data Validation | Resolved

Code Section

- contracts/config/assetManagement/PortfolioSettings.sol#L70-L72

Description

The variables `transferable` and `transferableToPublic` are not validated during the contract deployment and may lead to bricking the smart contract with incorrect initialization values, requiring it to be deployed again in order to be fixed. This is due to the validations being made on the function `updateTransferability()`.

Recommendation

It is recommended to perform the same validations during contract deployment as the ones being made on the function `updateTransferability()` in order to maintain the consistency of the storage values.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Incorrect Usage Of Initializing Functions

Low | **RES-VLVT-V3C09** | Code Quality | **Acknowledged**

Code Section

Not specified.

Description

The functions `_init()` and `_init_unchained()` are an implementation of OpenZeppelin's Upgradeable Proxies design standard and are used to serve as the constructor of upgradeable contracts. Despite being both used as initializers, they possess slightly different use cases to correctly implement the analog linearization of smart contract constructors.

The function `_init()` should be used and implemented to embed the linearized calls to all parent initializers. The function `_init_unchained()` should be used to perform the initialization of the variables for the current contract only.

As a consequence of this design, it is possible that, due to the lack of automatic linearization, the calling of `_init()` functions initializes the same contract multiple times. For these cases, the `_init_unchained()` function may be used manually to avoid double initialization, however it will break the upgradeability design and compatibility between smart contracts on the blockchain.

Additionally, the inherited smart contracts should be initialized according to their inheritance order, from the most base-like to the most derived.

There are multiple instances where contracts do not follow the design standard in the following account:

- `_init_unchained()` function is not used.
- Initialization according to inheritance order is incorrect
- Missing initialization of certain contracts

Recommendation

It is recommended to follow OpenZeppelin's Upgradeable Proxies design standard to the best of the possibilities, and follow the necessary recommendations to maintain composability, interoperability, and consistency across the blockchain.

It should be noted however, that it is not always possible to follow the recommendations to their full extend due to double initialization issues, and in those cases it may be required to manually initialize the parent contracts.

Status

The issue was acknowledged by Velvet's team. The development team stated "Inheritance order and missing initialization issues have been fixed. The implementation of init*unchained() functions has been deferred.".*

Potentially Invalid Logic In Role Transfer Function

Low | **RES-VLVT-V3C10** | Data Validation | **Acknowledged**

Code Section

- contracts/access/AccessController.sol#L74-80

Description

The `AccessController` contract defines a `transferSuperAdminOwnership`. This function is responsible for revoking the `SUPER_ADMIN` role from the previous holder and assigning it to a new user. The function internally calls the `revokeRole` and `_setupRole` and functions from the OpenZeppelin's `AccessControl` contract. Both of these functions execute their logic only if the a given `address` holds the role or does not respectively. As a consequence, it was observed that two specific scenarios are possible:

1. The supposed previous role holder does not actually hold the role. In this case, the `revokeRole` function will do nothing, while `_setupRole` function will assign the `SUPER_ADMIN` role to a new `address`. This scenario leads to potentially unwittingly creating new `SUPER_ADMIN` users without revoking previous role owners.
2. If the `transferSuperAdminOwnership` function is called with the same address provided as a `_oldAccount` and `_newAccount` parameters, first the code will attempt to assign new role to this user. This action will be effectively a no-op, as this user already holds this role. However, right after that, the `revokeRole` function will revoke the `SUPER_ADMIN` privileges from this user. As a consequence, the protocol would be left with no `SUPER_ADMIN`.

Recommendation

It is recommended to implement a verification mechanism that will make sure supposed previous role holder actually has that role. The execution of the function should only be continued if that is the case. Additionally, a check assuring that the arguments provided to the `transferSuperAdminOwnership` function are different should be implemented.

Status

*The issue was acknowledged by Velvet's team. The development team stated "In the contract "AccessController," only the admin can call the function transferSuperAdminOwnership. In this case, the admin is the DEFAULT*ADMIN*ROLE, which is only granted to the PortfolioFactory contract. In the PortfolioFactory, we have an external function transferSuperAdminOwnership which can only be called by a SUPER_ADMIN. Only from this function can the super admin role be transferred.".*

Excessive Fee Configuration is Possible for Protocol Fee Settings

Low | **RES-VLVT-V3C11** | Governance | **Resolved**

Code Section

- contracts/config/protocol/ProtocolFeeManagement.sol#L36-L48

Description

This vulnerability arises in the functions `updateProtocolFee` and `updateProtocolStreamingFee` of the smart contract `contracts/config/protocol/ProtocolFeeManagement.sol`:

```
ftrace | funcSig
function updateProtocolFee(uint256 _newProtocolFee⬆) external onlyProtocolOwner {
    protocolFee = _newProtocolFee⬆;
    emit ProtocolFeeUpdated(_newProtocolFee⬆);
}

ftrace | funcSig
function updateProtocolStreamingFee(uint256 _newProtocolStreamingFee⬆) external onlyProtocolOwner {
    protocolStreamingFee = _newProtocolStreamingFee⬆;
    emit ProtocolStreamingFeeUpdated(_newProtocolStreamingFee⬆);
}
```

In current actual shape, the function permits to update protocol fees to 100% by the protocol Owner, allowing potentially malicious contract owners or authorized users to configure fees that could seize the entirety of a transaction's value. This can lead to fraudulent revenue generation, severely impacting user trust and the economic stability of the platform.

If an attacker gains control over the contract owner's account (through phishing, private key compromise, etc.) and sets the transaction and protocol fees to 100%. Consequently, every transaction performed on the platform results in users losing all their transferred funds to fees, effectively rendering the platform unusable and draining user assets. This not only disrupts service but also destroys user confidence and could potentially lead to significant financial losses for all stakeholders involved.

Recommendation

To mitigate this vulnerability, it is crucial to set a fee limits through robust validation mechanisms within the smart contract. Implement a fee cap within the contract's fee-setting functions to prevent the configuration of disproportionately high fees. For example, establish a maximum allowable fee percentage (e.g., 10%) that aligns with the project's economic model and preserves overall functionality and user experience:

```
  ...

abstract contract ProtocolFeeManagement is OwnableCheck {
  uint256 public constant MAXIMUM_FEES_AMOUNT = 1000; // Maximum allowable fee set
↪   to 10%
  ...
```

```
  function updateProtocolFee(uint256 _newProtocolFee) external onlyProtocolOwner {
    require(_newProtocolFee <= MAXIMUM_FEES_AMOUNT, "Fee exceeds maximum allowed
↪   limit.");
    protocolFee = _newProtocolFee;
    emit ProtocolFeeUpdated(_newProtocolFee);
  }

  function updateProtocolStreamingFee(uint256 _newProtocolStreamingFee) external
↪   onlyProtocolOwner {
    require(_newProtocolStreamingFee <= MAXIMUM_FEES_AMOUNT, "Streaming fee exceeds
↪   maximum allowed limit.");
    protocolStreamingFee = _newProtocolStreamingFee;
    emit ProtocolStreamingFeeUpdated(_newProtocolStreamingFee);
  }
}
```

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Lack of ERC165 Checks

Info | **RES-VLVT-V3C12** | Data Validation | **Acknowledged**

Code Section

- contracts/config/assetManagement/PortfolioSettings.sol#L54
- contracts/config/assetManagement/TreasuryManagement.sol#L26
- contracts/config/assetManagement/TreasuryManagement.sol#L41
- contracts/config/protocol/OracleManagement.sol#L38
- contracts/config/protocol/ProtocolTreasuryManagement.sol#L39
- contracts/config/protocol/TokenManagement.sol#L32
- contracts/core/access/AccessModifiers.sol#L23

Description

The `ERC165` standard defines a `supportsInterface` function which is designed to verify that a particular `address` has a contract deployed that implements expected functionalities. Using `ERC165` checks is considered a good practice as it eliminates most of the user errors related to providing invalid addresses. Furthermore, the `supportsInterface` check also indirectly assures that no zero-address was provided.

Recommendation

It is recommended to implement `ERC165` standard and leverage the `supportsInterface` function it provides to assure an `address` used for cross-contract calls implements expected functionalities.

Status

The issue was acknowledged by Velvet's team. The development team stated "We have a script initializing those contracts and while testing we would realize if the address is wrong. Also we have a function to update the oracle address in case any wrong address is passed during the setup. To avoid the case of initializing the contracts with wrong addresses we can also add checks on the front-end".

Redundant Checks On latestRoundData()

Info | **RES-VLVT-V3C13** | Gas Optimization | **Resolved**

Code Section

- contracts/oracle/PriceOracle.sol#L36-L44

Description

The `PriceOracle` contract implements a `latestRoundData` function responsible for returning the most recent price. This function executes various checks, making sure that the price is accurate, i.e. that it is not equal to zero and that it is not expired. It was observed that the expire check is done twice - via a `require` statement and then as a manual `if` statement. Having two checks that verify the same thing is redundant. If the check would cause the execution to fail, only one statement will suffice to cause this. On the other hand, if the price is not expired, then the contract will effectively execute the same check twice, making it less gas efficient.

Recommendation

It is recommended to remove unnecessary or redundant code in order to save on gas fees and to make the code more readable.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Unused Functions

Info | **RES-VLVT-V3C14** | Code Quality | **Resolved**

Code Section

- contracts/core/config/VaultConfig.sol#L127-L131
- contracts/fee/FeeConfig.sol#L66-L71

Description

The following functions and modifiers were found to be unused within the system:

- `_lastSnapshotId()`
- `notPaused()`

Unused functions increase the complexity and readability of the smart contract's code and their inclusion should be discouraged whenever possible.

Recommendation

It is recommended to remove unused functionalities from production-ready code.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Price Check Not Taking Into Account The Return Type

Info | **RES-VLVT-V3C15** | Code Quality | **Resolved**

Code Section

- contracts/oracle/PriceOracle.sol#L46

Description

The `PriceOracle` contract responsible for fetching prices from Chainlink's oracles implements some checks to make sure that the price returned by the Chainlink oracle is valid. One of those checks is to ensure that it is not equal to `0`. However, Chainlink oracles contain variety of information, not necessarily the price of an asset with regards to another asset. This is why, the `answer` returned by Chainlink's oracle is of type `int256`. Although it is very unlikely for an asset price oracle to return a negative price, the check if it is equal to `0` would not catch this case.

Recommendation

It is recommended to modify the check for an invalid price so that it ensures the `answer` is positive, instead of just not being equal to `0`.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Unused Input Parameter _protocolConfig

Info | RES-VLVT-V3C16 | Code Quality | Resolved

Code Section

- contracts/FunctionParameters.sol#L145
- contracts/access/AccessController.sol#L46-L67

Description

The following variables were found to be unused within the system:

- `_protocolConfig` in `setUpRoles()`

Unused variables increase the complexity and readability of the smart contract's code and their inclusion should be discouraged whenever possible.

Recommendation

It is recommended to remove unused variables from production-ready code.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Redundant Checks On updateTokens()

Info | RES-VLVT-V3C17 | Gas Optimization | Resolved

Code Section

- contracts/rebalance/Rebalancing.sol#L120-L123
- contracts/rebalance/Rebalancing.sol#L148
- contracts/core/config/VaultConfig.sol#L97

Description

The function `updateTokens()` indirectly performs the same validations provided by `beforeInitCheck()` twice on the same variable `_newTokens`. The first time occurs within `_updateTokensCheck()` where every token is verified in a loop and performs a call to `beforeInitCheck()`. The second time occurs right after on `updateTokenList()` that does the exact same validations in a for loop.

Recommendation

It is recommended to remove unnecessary or redundant code in order to save on gas fees and to make the code more readable.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Redundant Checks On multiTokenWithdrawal()

Info | **RES-VLVT-V3C18** | Gas Optimization | **Resolved**

Code Section

- contracts/core/management/VaultManager.sol#L95-L101
- contracts/core/management/VaultManager.sol#L263
- contracts/core/checks/ChecksAndValidations.sol#L68

Description

The function `multiTokenWithdrawal()` indirectly performs the same validations of the user's portfolio tokens for withdrawal twice. The first time occurs within `_beforeWithdrawCheck()` and the second occurs right after on `_validateUserWithdrawal()`. On both instances, if failing, the checks revert the transaction with the same error `CallerNotHavingGivenPortfolioTokenAmount()`.

Recommendation

It is recommended to remove unnecessary or redundant code in order to save on gas fees and to make the code more readable.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Transfer Of Super Admin Ownership Does Not Revoke Other Roles

Info	RES-VLVT-V3C19	Access Control	Acknowledged

Code Section

- contracts/access/AccessController.sol#L74-L80

Description

The function `transferSuperAdminOwnership()` is used to transfer the `SUPER_ADMIN` role from one account to another. When successful, the account of the address identified by the variable `_newAccount` will be the new `SUPER_ADMIN`, and the variable `_oldAccount` will identify the account that was stripped of this role. However, it should be noted that only the ownership of this role is transferred, which means that, previous roles that may have been assigned by the previous `SUPER_ADMIN` to themselves or others will stay in place.

In cases the `SUPER_ADMIN` role is being transferred due to a compromise of the respective account, this issue becomes more relevant and impactful as the malicious actor may assign important roles on the protocol that will serve as a backdoor for future use.

Recommendation

It is recommended to verify and guarantee that the necessary roles are updated and/or revoked whenever a transfer of the `SUPER_ADMIN` role occurs.

Status

The issue was acknowledged by Velvet's team. The development team stated "We are aware that the account might still have other roles, but this function is intended to transfer the super admin, who can then revoke all other roles as an admin.".

Missing Length Checks

Info | **RES-VLVT-V3C20** | Data Validation | **Resolved**

Code Section

- contracts/handler/ExternalSwapHandler/EnsoHandlerBundled.sol#L42
- contracts/rebalance/Rebalancing.sol#L99

Description

The Velvet Capital contracts are often using arrays as parameters for their functions. Those arrays are usually looped over to execute certain actions. However, it was observed that in some cases, there checks to make sure that the arrays necessary for a given execution flow have matching lengths. As a result, it is possible to provide arrays with different lengths that will make the execution fail when EVM will try to access an element that does not exist in an array.

The missing checks were identified for contracts:

- `EnsoHandlerBundled` - missing check on `tokensLength` and `callDataEnso.length` in `multiTokenSwapAndTransfer` function.
- `Rebalancing` - missing check on `sellAmounts.length` and `sellTokens.length` in `_updateWeights` function.

Recommendation

It is recommended to add a length checks to prevent scenarios where a loop would start executing on an invalid input data.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Possible Accidental Event Flooding

Info | RES-VLVT-V3C21 | Code Quality | Resolved

Code Section

- contracts/config/assetManagement/FeeManagement.sol#L114
- contracts/config/assetManagement/FeeManagement.sol#L153
- contracts/config/assetManagement/FeeManagement.sol#L199

Description

The `FeeManagement` contract is responsible for holding values for various fees associated with using the protocol. It also implements a functionality to change values of those fees. The update mechanism is a 2-step process that uses a time period until the second step, actual fee change, can be executed. Using such a time period is considered a good practice. However, it was observed that after one successful update, the same update function can be called indefinitely, until new fee proposal is created. Such subsequent update calls wouldn't change the fee, however they would still emit an event which might disrupt operation of any entity that uses this data.

This finding is related to the `updateManagementFee`, `updatePerformanceFee` and `updateEntryAndExitFee` functions.

Recommendation

It is recommended to change the proposed fee times to `0` after the fee was changed, so that subsequent calls would result in a `NoNewFeeSet` error.

Status

The issue has been fixed in 52281bcef61f6b4e6e58bb6e7e46bc71be3d8d22.

Proof of Concepts

No Proof-of-Concept was deemed relevant to describe findings in this engagement.